AD-A054 505   CALIFORNIA UNIV SANTA CRUZ INFORMATION SCIENCES   F/G 5/7

A CONCISE EXTENSIBLE METALANGUAGE FOR TRANSLATOR IMPLEMENTATION--ETC(U)

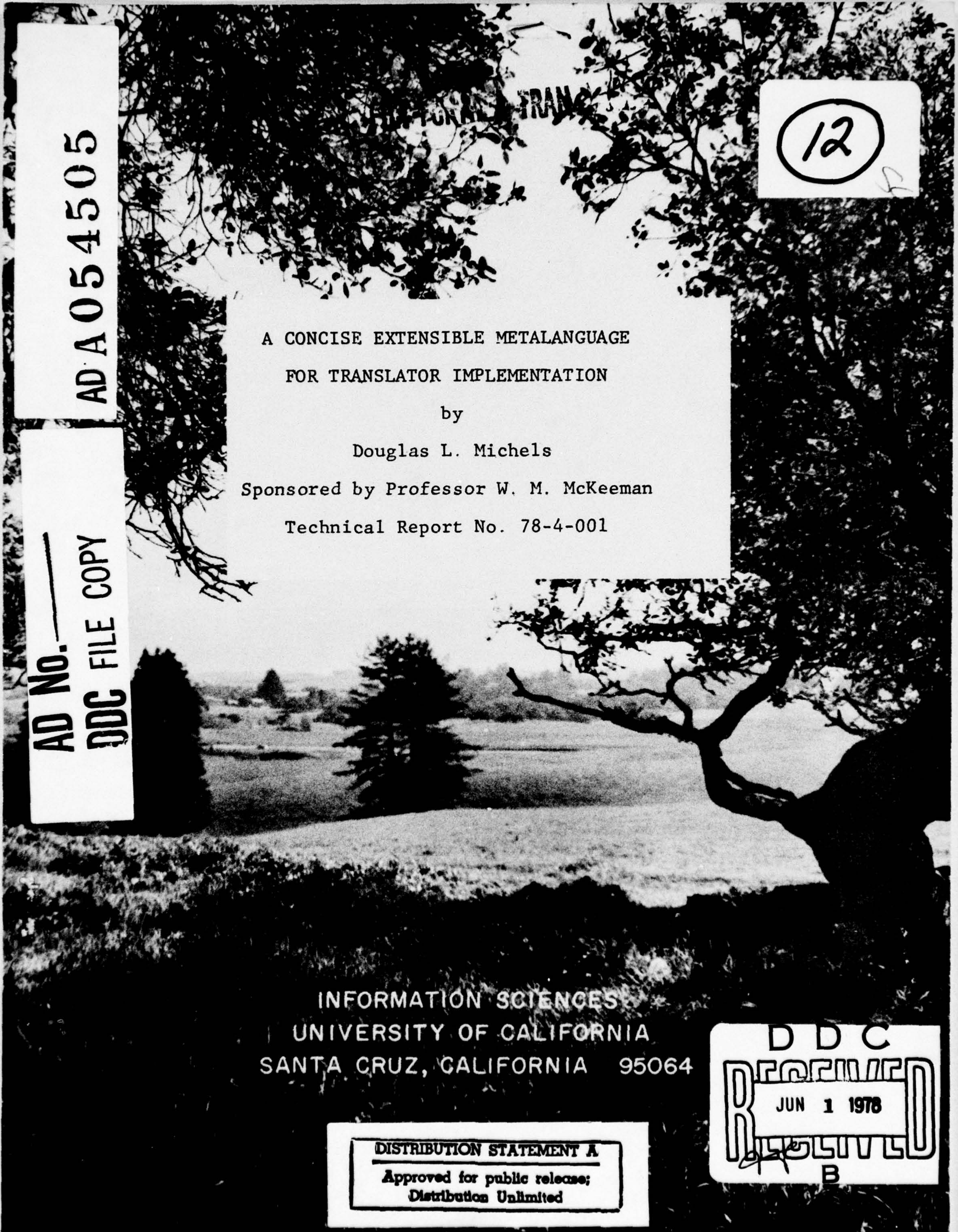
JUL 76 D L MICHELS   N00014-76-C-0682

UNCLASSIFIED   TR-78-4-001   NL

1 OF 1

AD
A054505

END
DATE
FILMED
6-78
DDC



A CONCISE EXTENSIBLE METALANGUAGE

FOR TRANSLATOR IMPLEMENTATION

by

Douglas L. Michels

Sponsored by Professor W. M. McKeeman

Technical Report No. 78-4-001

INFORMATION SCIENCES
UNIVERSITY OF CALIFORNIA
SANTA CRUZ, CALIFORNIA 95064

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: 14 TR-78-4-001	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): 6 A CONCISE EXTENSIBLE METALANGUAGE FOR TRANSLATOR IMPLEMENTATION.		5. TYPE OF REPORT & PERIOD COVERED: 9 Technical rept.
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): 10 Douglas L./Michels		8. CONTRACT OR GRANT NUMBER(s): ONR CONTRACT NO. ONR N00014-76-C-0682 15
9. PERFORMING ORGANIZATION NAME AND ADDRESS: William M. McKeeman and Sharon/Sickel, Information Sciences, UCSC, Rm. 239 Applied Sciences, Santa Cruz, Ca. 95064		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 11 25 Jul 76
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research, Arlington, Virginia 22217		12. REPORT DATE: July 25, 1976
		13. NUMBER OF PAGES: 28 12 32p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research, University of California, 553 Evans Hall, Berkeley, California 94720		15. SECURITY CLASS. (of this report): Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce, for sale to the general public.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Translator, compiler, translator writing system, metalanguage, metacompiler, self-describing grammar, interpreter

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A class of emitter augmented phrase structure grammars is defined which can specify simple translations of context free languages. A self-translating metatranslator for the description of these translation grammars is described. Several mutually recursive functions define an interpreter which will execute grammars as translated by this metatranslator.

(continued on next page)

410 350

JOB

20. ABSTRACT (continued)

The evolution of more sophisiticated translations systems is discussed and extensions to the metatranslator and interpreter are demonstrated.

A very concise self-describing metalanguage and an interpreter which will directly execute its self-description are presented.

(1473B)

A CONCISE EXTENSIBLE METALANGUAGE FOR TRANSLATOR IMPLEMENTATION

Douglas L. Michels

Information Sciences
University of California
at
Santa Cruz

July 25, 1976

This research was supported partially by Office of Naval Research Contract No. N00014-76-C-0682.

A CONCISE EXTENSIBLE METALANGUAGE FOR TRANSLATOR IMPLEMENTATION

Douglas L. Michels

Information Sciences
University of California
at
Santa Cruz

July 25, 1976

ABSTRACT

A class of emitter augmented phrase structure grammars is defined which can specify simple translations of context free languages. A self-translating metatranslator for the description of these translation grammars is described. Several mutually recursive functions define an interpreter which will execute grammars as translated by this metatranslator.

The evolution of more sophisticated translations systems is discussed and extensions to the metatranslator and interpreter are demonstrated.

A very concise self-describing metalanguage and an interpreter which will directly execute its self-description are presented.

Key words and phrases:

Translator, Compiler, Translator Writing System, Metalanguage, Metacompiler, Self-describing grammar, Interpreter

CR catagories: 4.12, 4.13, 4.20

1. Introduction

A LANGUAGE is a set of strings. A METALANGUAGE is a language for the description of languages. A self-describing metalanguage is a language which can express its own description. A RECOGNIZER for a language is a boolean function that when applied to a string is true if and only if the string is a member of the language. A translation language is a metalanguage that can express the mapping of a source to an object language. A translation language implicitly defines both the source and object language. A TRANSLATOR is the implementation of translation mapping. If a null object language is produced by a translator then it is equivalent to a recognizer. A METATRANSLATOR is a translator that maps a translation language to an object language. A metatranslator that can be described with its own source language is self-translating. The object language produced by a translator may be in any form; if it is the machine language of some computer then the translator is a COMPILER for that computer. If the object language directs the execution of a computer program it is an interpretive language and the program it controls is an INTERPRETER. An EXTENSIBLE metatranslator is one that can describe translators that have capabilities it does not itself have.

A very simple self-translating metatranslation language can be used to evolve arbitrarily sophisticated metatranslators. The first step towards accomplishing this is to define the simplest such language, extensible enough to provide for future evolution. This report is an attempt to define one such initial metalanguage and discuss its future evolution. A brief history of such systems is presented in section 2. In section 3 a class of translators, adequate for an initial metalanguage, is formally defined. A metatranslator that maps the description of such translators to an executable object program is described in section 4.1. This description is in the form of the source of language it defines; the self-translation of this description is also presented. Section 4.2 describes the metatranslator's object language by presenting a recognizer for it in the

metatranslator source language. The object language translation of this recognizer is also presented. An interpreter that will execute programs in the metatranslator object language is functionally defined in section 4.3.

An extended metatranslator can then be created by using this initial one. A description of the extended metatranslator written in the source language of the original metatranslator is given in section 5.1. The translation of this description is executable on the interpreter for the original object language. This new metatranslator accepts an extended source language and produces an extended object language. A recognizer for the extended object language, described with the extended source language, and the translation of this recognizer to the extended object language are provided in section 5.2. The extensions to the interpreter necessary to execute this extended object language are presented in section 5.3.

In section 6.1 it is demonstrated that a simplified version of the object languages produced by these metatranslators is itself a very simple self-describing metalanguage. A simple interpreter for this source/object language is defined in section 6.3.

2. History

Traditionally, computer instructions have been poorly suited to the expression of solutions to many human problems. To facilitate the use of computers, programming languages which allow a more problem-oriented statement of a solution have been created to provide an interface between human and machine languages. A translator is a computer program that can translate instructions in some specified programming language to an equivalent program in some other form.

The creation of a translator in a machine language or a general purpose programming language is a difficult and error prone process. A system which can automatically create a compiler from some concise description is called a TRANSLATOR WRITING

SYSTEM (TWS) and many approaches to designing such systems have been proposed.

Formal language theory has provided convenient techniques for the definition of a language. Finite grammars can be used to specify an infinite set of strings which comprise a specific language [Chomsky 57]. Algorithms to generate efficiently implementable recognizers from the grammar for a specific language have been discovered for several useful classes of languages [DeRemer 71, Knuth 65, Floyd 63].

Several methods have been used to extend these recognition techniques for use in translation. The recognizer can maintain a history of the order in which productions of the grammar are applied, resulting in a canonical parse [Wirth 66]. The production system can be extended to include an optional transduction rule associated with each production. These rules are applied in parallel with associated productions, resulting in an Abstract Syntax Tree [Wozencraft 65]. Both of these methods result in a representation of the source program which is then refined by a program in some programming language. Another alternative is to augment grammar productions with output strings that specify the strings to be emitted. It has been demonstrated that for several useful languages this method is capable of directly generating translations in the form of an assembly language program for a language specific interpreter [Schorre 64].

McKeeman [76] has suggested a refinement approach to the construction of translator writing systems. This approach is based on partitioning the system into several languages, one for each major component of the resultant translators. Instead of constructing the TWS in its totality, it is to be "evolved", each generation a product of the tools created by the previous generation. The design objective for each generation is the creation of the most useful tools with which to "evolve" the next generation.

This evolution must begin somewhere. McKeeman [76] has named this basis step a SEED. The seeds of a translator writing system are the tools necessary to create minimal versions of sufficient translator description languages to describe more sophisticated

languages. An ideal seed would have the capability to build several very simple but significantly different translators. The seed and the languages constructed with it serve only as development steps and therefore no optimizations other than conceptual clarity and extensibility need be considered.

3. Formal Definition of Translation

The translation mapping of a context-free source language L to an object language O can be generated by a context-free translation grammar T. If G = (V_t, V_n, S, P) is a context-free grammar generating L then the translation grammar T = (V_t, V_n, V_d, S, P'), where:

V_t is a finite set of symbols called TERMINALS.
V_n is a finite set of symbols called NON-TERMINALS.
V_d is a finite set of symbols called OUTPUTS, or DESTINATIONS.
V_t, V_n and V_d are mutually disjoint.
is the union of V_t, V_n and V_d and is called the ALPHABET.
S is a distinguished member of V_n called the start or goal symbol.
P is a finite set of productions such that each production is a pair (a,b). The LEFT PART a is a symbol in V_n and the RIGHT PART b is a sequence of symbols from the union of V_t and V_n.
P' is a finite set of productions such that each production is a pair (a,b). The left part a is a symbol in V_n and the right part b is a sequence of symbols from the union of V_t, V_n and V_d.

The postfix operator * will denote the set closure or the set of all sequences of symbols in a set. For example, V* represents the set of all strings that can be constructed from the symbols in the alphabet, including the empty string. The operator + denotes the set closure with the exclusion of the empty string.

The set of productions define all possible derivations in T. For all (a,b) in P' and u,v in V*, u is derivable from v if u can

be created by the substitution of b for any occurrence of a in v, or in any derivation of v.

Any sequence of symbols derivable from S is a SENTENTIAL FORM. A sentential form not containing any elements of V_n is a FINAL SENTENTIAL FORM, that is, no further derivation is possible. The deletion of all elements in V_d from a final sentential form will produce a TERMINAL SENTENCE and all such sentences are in the language L. The deletion of all elements in V_t from a final sentential form will produce an OUTPUT SENTENCE and all such sentences are in the object language defined by T. An output sentence is the translation by T of a terminal sentence if there exists a sequence in final sentential form from which both the output and terminal sentence can be produced.

3.1 A Translation Example

A grammar that will translate infix expressions to prefix is shown as an example of this class of translation. A translator M can be described by $T = (V_t, V_n, V_d, S, P')$ where:

$V_t = \{+,*,a,b\}$ This is the alphabet of the source language.

$V_n = \{S,T,F,I\}$ These are the non-terminal symbols.

$V_d = \{P,X,A,B\}$ This is the object language alphabet; in this case it corresponds one for one with V_t. The symbols were renamed to differentiate the two sets.

$P' = \{(S,T),(T,PF+T),(T,F),(F,XI*F),(F,I),(I,Aa),(I,Bb)\}$ This is the set of productions defining the translation.

M defines the mapping of 'a+b*a' to '+a*ba'. The full derivation is as follows:

Sentential Form	Transitional Rule	
S	Start symbol	
T	(S,T)	
PF +T	(T,PF+T)	
PF +F	(T,F)	
PF +XI *F	(F,XI*F)	
PI +XI *I	(F,I)	
PA a+XB b*A a	(I,A a),(I,B b)	Final Sentential Form
a+ b* a	Terminal Sentence	Delete all symbols from V_d
PA XB A	Output Sentence	Delete all symbols from V_t
+a *b a	Output Sentence mapped back to the corresponding input vocabulary	

4. Translator Implementation

A language for describing translators of the type just defined can be created such that only the set of productions need be stated. To do this the language must provide a way to differentiate the symbols of each vocabulary. Each vocabulary is then defined to contain only those symbols denoted in the productions. Certain assumptions, restrictions and conventions can be asserted to greatly facilitate a top down, deterministic implementation of such translator descriptions.

No productions may be empty. That is, for all (a,b) in P every b must be a member of V+.

PL is constructed from P, the set of productions. Each element of PL is a list $(a,b_1,b_2,\ldots,b_n)$; a is some left part and all b_i are corresponding right parts. That is, all b_i are included in an element of PL if and only if (a,b_i) is an element of P. A translator will be represented by a description of PL, such that the first element is the list in which a = S, the start symbol.

An ordering on the alternative right parts in each element PL is defined to guarantee that if two possible derivations have terminal sentences, such that one is a right substring of the other, the longer will be listed first. For all l in in PL

and b, b' in V+, if b, b' are right parts in 1 and b precedes b' then for all u, u' elements of V_t+ such that u and u' are derivable from b and b' respectively there exists no u' = ur where r is in V*.

Languages cannot be specified which allow left recursion. This would result in an infinite recursion in a top-down left to right parse. If for any v in V_n and any u in V*, vu is derivable from v then the grammar allows left recursion.

A production is capable of deriving an arbitrary number of repetitions of a particular terminal sequence. If the specification of a terminal sequence follows a specificiation allowing an arbitrary repetition of the same sequence then there is no deterministic left to right parse. That is, for all productions (a,b) in P there must not exist any sequence umcv derivable from b, and cv derivable from m, where u, m, and v are in V* and c is in V_t+.

4.1 A Simple Translation Language

A Metalanguage, utilizing a limited character set, can be defined for the syntax and translation of an emitter augmented, phrase structure grammar. The notation is similar to BNF [Naur 60], however terminals and output symbols are quoted with non-terminals being single characters. The vertical bar (|) will be used to separate alternative right parts in an element of PL. The left part will be separated from the alternative right parts by an equal sign (=). Juxtaposition will denote the concatenation of definitions. Single quotes (') will delimit elements in V_t+. Brackets ([,]) will be used to delimit elements of V_d+. Normal parentheses can be used to alter the implied operator precedence and to reduce the number of productions required by allowing the factoring of rules.

Literal strings may be of arbitrary length. This creates a problem if a string must contain a single quote ('), which is the literal delimiter. To solve this the production 'I' is ordered to test for a single quote as the first character of a literal string; if one is found it is assumed to be the entire

string and must be followed by the terminating single quote. A single quote in any other position of a literal string is assumed to be the terminating delimiter of that string.

The infix to prefix translator of section 3.1 is rewritten in the syntax of this metatranslator as an example. The terminal and output vocabularies are the same as they can be differentiated by their delimiters.

```
BEGIN GRAMMAR
T = [+] F '+' T
  | F
  ;
F = [*] I '*' F
  | I
  ;
I = [a] 'a'
  | [b] 'b'
  ;
END GRAMMAR
```

The translation language describing this notation and its translation is expressed in its own language. Multiple blanks and end of lines have no meaning in the language. They have been used here to improve readability and should be ignored. A realistic treatment of multiple blanks is demonstrated in the translation language of section 5.1.

```
BEGIN GRAMMAR
G = 'BEGIN GRAMMAR' R 'END GRAMMAR'
R = L '=' A ';' R
  | L '=' A ';'
  ;
A = [|] C '|' A
  | C
  ;
C = [&] I ' ' C
  | I
  ;
I = ''' ( ''' ["'] | S ) '''
  | '[' ( O | ']' [>] []] ) ']'
```

```
    | '(' A ')'
    | [:] L
    ;
S   = [&"] ( L | ']' []] ) S
    | ["] ( L | ']' []] )
    ;
O   = [&>] ( L | ''' ['] ) O
    | [>] ( L | ''' ['] )
    ;
L   = 'A' [A] | 'B' [B] | 'C' [C] | 'D' [D] | 'E' [E] | 'F' [F]
    | 'G' [G] | 'H' [H] | 'I' [I] | 'J' [J] | 'K' [K] | 'L' [L]
    | 'M' [M] | 'N' [N] | 'O' [O] | 'P' [P] | 'Q' [Q] | 'R' [R]
    | 'S' [S] | 'T' [T] | 'U' [U] | 'V' [V] | 'W' [W] | 'X' [X]
    | 'Y' [Y] | 'Z' [Z] | '=' [=] | ';' [;] | '(' [(] | ')' [)]
    | '|' [|] | '&' [&] | '>' [>] | ':' [:] | '"' ["] | ' ' [ ]
    | '[' [[] | '*' [*]
    ;
END GRAMMAR
```

The self-translation of the above translator description is as follows:
(Paragraphing has been added to improve readability. The actual machine language, as defined by the translator and accepted by the machine, would be a continuous string of characters. The only significant blank is one which follows a " .)

```
G  &&"B&"E&"G&"I&"N&" &"G&"R&"M&"M&"A"R
     &:R
      &"E&"N&"D&" &"G&"R&"A&"M&"M&"A"R
R  |&:L
     &"=
       &:A
         &";
           :R
   &:L
     &"=
       &:A
         ";
A  |&>|
     &:C
       &"|
         :A
   :C
C  |&>&
     &:I
       &"
         :C
   :I
```

```
I  |&"'
    &|&"'
      &>"
       >'
     :S
     "'
   |&"[
     &|:O
      &"]
       &>>
        >]
      "]
    |&"(
     &:A
      ")
     &>:
      :L
S  |&&>&
     >"
    &|:L
     &"]
      >]
     :S
   &>"
    |:L
     &"]
      >]
O  |&&>&
     >>
    &|:L
     &"'
      >'
     :O
   &>>
    |:L
     &"'
     >'
L  |"A>A|"B>B|"C>C|"D>D|"E>E|"F>F|"G>G|"H>H|"I>I|"J>J|"K>K|"L>L
   |"M>M|"N>N|"O>O|"P>P|"Q>Q|"R>R|"S>S|"T>T|"U>U|"V>V|"W>W|"X>X
   |"Y>Y|"Z>Z|"=>=|";>;|"(>(|")>)|"|>||"&>&|">>>|":>:|"">"|" >
   |"[>[ "*>*
```

4.2 The Object Language

The output of the metatranslator described by section 4.1 is a program executable by the interpreter defined in section 4.3. This object language is a context-free language and a recognizer for it can be described in the syntax of the section 4.1 metatranslator.

```
BEGIN GRAMMAR
G = L R G
  | L R
  ;
R = ':' L
  | '&' R R
  | '|' R R
  | '>' L
  | ''' L
  ;
L = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K'
  | 'L'   'M'   'N'   'O'   'P'   'Q'   'R'   'S'   'T'   'U'   'V'
  | 'W'   'X'   'Y'   'Z'   '&'   '|'   ':'   '''   '>'   '('   ')'
  | '''   ' '   '['   ']'   '='   ';'   '*'
  ;
END GRAMMAR
```

The object language as described contains five prefix operators. The '&' is a binary concatenation operator, it has the value TRUE only if both of the operands which follow it are true. The '|' is the binary alternation operator, it has the value TRUE if either of the operands following it are TRUE. If the first is TRUE the second is not tested. If the first is FALSE both input and output strings are restored to their pre-test value before testing the second. The ':' is a unary non-terminal operator; it has the value TRUE if the rule labeled by its operand is TRUE. The ''' is a unary terminal operator; it has the value TRUE if the current character of input is the same as its operand and the input is advanced one character. The '>' is a unary operator and always has the value TRUE. The character following it is appended to the right of the current output string.

A translation of the above grammar according to the meta-translator of section 4.1 would be:
(Again paragraphing has been used to improve readability.)

```
G |&:L
    &:R
      :C
   &:L
     :R
R |&":
     :L
   |&"&
      &:R
        :R
    |&"|
        &:R
          :R
     |&">
          :L
        &""
          :L
L |"A|"B|"C|"D|"E|"F|"G|"H|"I|"J|"K|"L|"M|"N|"O|"P|"Q|"R|"S|"T|"U
  |"V|"W|"X|"Y|"Z|"&|"||":|""|">|"(|")|"'|" |"[|"]|"=|";"*
```

4.3 The Translator Interpreter

An interpreter that will execute the object language of section 4.2 and perform translations as formalized can be defined in terms of several mutually recursive functions.

The object language program, the input, and the output can each be considered to be a finite sequence of characters or a STRING. To facilitate the definition of the machine, some primitive operations on strings will be defined.

String Operations:

First: STRING -> STRING
  First (S) is the single left-most character of S.

Rest: STRING -> STRING
  All but the left-most character of the string.

Concat: STRING × STRING -> STRING
  S = concat (first(S),rest(S))

Equal: STRING x STRING -> BOOLEAN

Equal (S1,S2) is TRUE if and only if S1 is identical to S2.

Three sets of strings are of interest:

STRING(i) - Strings of object code executable by an interpreter.

STRING(s) - Strings in the source language of a translator defined by an element of STRING(i).

STRING(o) - Strings in the object language produced by a translator defined by an element of STRING(i).

Functionality and function of interpreter definition functions:

Machine: STRING(i) x STRING(s) -> BOOLEAN x STRING(o)

(RECOGNIZE,OUTPUT) = Machine (GRAMMAR,INPUT)

If INPUT is described by GRAMMAR then RECOGNIZE is TRUE and OUTPUT is the object program produced when GRAMMAR is applied to INPUT.

Test: STRING(i) x STRING(i) x STRING(s) -> BOOLEAN

Test (GRAMMAR,RULE,INPUT) is TRUE if any left-most substring of INPUT is recognized by RULE. RULE is always a substring of GRAMMAR.

Remaining: STRING(i) x STRING(i) x STRING(s) -> STRING(s)

Remaining (GRAMMAR,RULE, INPUT) is the substring of INPUT remaining after the substring recognized by RULE has been removed.

Emit: STRING(i) x STRING(i) x STRING(s) -> STRING(o)

Emit (GRAMMAR,RULE,INPUT) is the translation of the substring of INPUT recognized by RULE.

Skip: STRING(i) -> STRING(i)

Skip (RULE) is the substring of RULE remaining after the leftmost operator and its operands have been removed.

Find: STRING(i) x STRING(i) -> STRING(i)

Find (GRAMMAR,STRING) is the substring of GRAMMAR labeled by the first character of STRING.

A recursive definition of these functions:

```
Machine (G,I) =
  IF Test (G,rest(G),I) AND equal (Remaining(G,rest(G),I),NULL)
     THEN
        (TRUE,Emit (G,rest(G),I)
     ELSE
        (FALSE,NULL)
END Machine

Test (G,R,I) =
  CASE first (R) OF
   ':' : Test (G,Find(G,rest(R)),I)
   '&' : IF Test (G,rest(R),I)
           THEN
              Test (G,Skip(rest(R)),Remaining(G,rest(R),I))
           ELSE
              FALSE
   '|' : IF Test (G,rest(R),I)
           THEN
              TRUE
           ELSE
              Test (G,Skip(rest(R)),I)
   '>' : TRUE
   '"' : equal (first(rest(R)),first(I))
  END CASE
END Test

Remaining (G,R,I) =
  CASE first(R) OF
   ':' : Remaining (G,Find(G,rest(R)),I)
   '&' : Remaining (G,Skip(rest(R)),Remaining(G,rest(R),I))
   '|' : IF Test(G,rest(R),I)
           THEN
              Remaining(G,rest(R),I)
           ELSE
              Remaining(G,Skip(rest(R)),I)
   '>' : I
   '"' : rest(I)
  END CASE
END Remaining
```

```
Emit (G,R,I) =
  CASE first(R) OF
  ':' : Emit (G,Find(G,rest(R)),I)
  '&' : Concat(Emit(G,rest(R),I),Emit(G,Skip(rest(R)),Remaining
          (G,rest(R),I)))
  '|' : IF Test (G,rest(R),I)
           THEN
              Emit (G,rest(R),I)
           ELSE
              Emit (G,Skip(rest(R)),I)
  '>' : first(rest(R))
  '"' : NULL
  END CASE
END Emit

Skip (R) =
  CASE first (R) OF
  ':' : rest(rest(R))
  '&' : Skip(Skip(rest(R)))
  '|' : Skip(Skip(rest(R)))
  '>' : rest(rest(R))
  '"' : rest(rest(R))
  END CASE
END Skip

Find (G,R) =
  IF equal (first(G),first(R))
     THEN
        rest(G)
     ELSE
        Find (Skip(rest(G)),R)
END Find
```

5. Extensions to the Metatranslator

A metatranslator written in the language of section 4.1 can translate an extended translation language. This extended language will allow identifiers representing symbols in V_n to be of arbitrary length. It will also permit the use of the postfix operator '*' to indicate zero or more repetitions of the preceding rule.

This translator and the interpreter necessary to execute the programs it produces can be used to create translators for more interesting languages. They also serve as an example of using an existing metatranslator to evolve a more complex one.

5.1 The Extended Metatranslator

A description of an extended metatranslator is presented in the syntax of the metatranslator described in section 4.1. Multiple blanks are taken into consideration. The end of line is ignored.

```
BEGIN GRAMMAR
G = 'BEGIN GRAMMAR' R B 'END GRAMMAR' ;
R = D '=' A ';' R
  | D '=' A ';'
  ;
D = B S B
  | S B
  | S
  ;
A = [|] C '|' A
  | C
  ;
C = [&] K C
  | K
  ;
K = [*] N '*'
  | N
  ;
N = B I B
  | I B
  | I
  ;
I = ''' ( S | ''' ["'] ) '''
  | '(' A ')'
  | '[' ( O | '[' [>] []] ) ']'
  | [:] V
  ;
S = [&"] ( L | ']' []] | ' ' [ ] ) S
  | ["] ( L | ']' []] | ' ' [ ] )
  ;
O = [&"] ( L | ''' ['] | ' ' [ ] ) O
  | ["] ( L | ''' ['] | ' ' [ ] )
  ;
V = [&"] L V
  | ["] L
  ;
L = 'A' [A] | 'B' [B] | 'C' [C] | 'D' [D] | 'E' [E] | 'F' [F]
  | 'G' [G] | 'H' [H] | 'I' [I] | 'J' [J] | 'K' [K] | 'L' [L]
  | 'M' [M] | 'N' [N] | 'O' [O] | 'P' [P] | 'Q' [Q] | 'R' [R]
  | 'S' [S] | 'T' [T] | 'U' [U] | 'V' [V] | 'W' [W] | 'X' [X]
  | 'Y' [Y] | 'Z' [Z] | '=' [=] | ';' [;] | '(' [(] | ')' [)]
  | '|' [|] | '&' [&] | ':' [:] | '>' [>] | '"' ["] | '[' [[]
  | '*' [*]
  ;
```

```
B = ' ' B
  | ' '
  ;
END GRAMMAR
```

5.2 An Extended Object Language

The metatranslator just defined produces an extended object language. This is required to support the language extensions now defined. A recognizer for this object language, in the language of the translater defined in section 5.1, follows:

```
BEGIN GRAMMAR
GRAMMAR = STRING RULE GRAMMAR
        | STRING RULE
        ;
RULE    = ':' STRING
        | '&' RULE RULE
        | '|' RULE RULE
        | '>' LITERAL
        | '"' LITERAL
        | '*' RULE
        ;
STRING  = '&' LITERAL STRING
        | '"' LITERAL
        ;
LITERAL = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J'
        | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T'
        | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | '&' | '|' | ':' | '"'
        | '>' | '(' | ')' | ' ' | ''' | '[' | ']' | '_' | ';' | '*'
        ;
END GRAMMAR
```

There are two extensions in this object language. The first is the addition of the unary '*' operator. This operator always yields the value TRUE, it specifies that the rule on which it operates be applied to the input repeatedly until it becomes false. Both the input and output strings are restored to their values previous to the evaluation of the rule that yielded the value FALSE. The second extension is a change in the meaning of ':' operator. Instead of operating on single character names, the ':' will operate on strings which are defined with '&' and '"' operators.

The translation of the object language recognizer by the translator of section 4.1 is shown. The language recognized by this grammar is executable on the interpreter of section 5.3 (paragraphing has been added).

```
&"G&"R&"A&"M&"M&"A"R  |&:&"S&"T&"R&"I&"N"G
                        &:&"R&"U&"L"E
                         :&"G&"R&"A&"M&"M&"A"R
                       &:&"S&"T&"R&"I&"N"G
                         :&"R&"U&"L"E
&"R&"U&"L"E  |&":
                :&"S&"T&"R&"I&"N"G
                |&"&
                  &:&"R&"U&"L"E
                   :&"R&"U&"L"E
                  |&"|
                    &:&"R&"U&"L"E
                     :&"R&"U&"L"E
                    |&">
                      :&"S&"T&"R&"I&"N"G
                     |&""
                       :&"L&"I&"T&"E&"R&"A"L
                      &"*
                        :&"R&"U&"L"E
&"S&"T&"R&"I&"N"G  |&"&
                      &:&"S&"T&"R&"I&"N"G
                       :&"S&"T&"R&"I&"N"G
                    &""
                      :&"L&"I&"T&"E&"R&"A"L
&"L&"I&"T&"E&"R&"A"L  |"A|"B|"C|"D|"E|"F|"G|"H|"I|"J|"K|"L|"M|"N
                        |"O|"P|"Q|"R|"S|"T|"U|"V|"W|"X|"Y|"Z|"&
                         |"||":|""|">|"(|")|" |""|"[|"]|"=|";"*
```

As this object language is not compatible with the previous object language, it is appropriate to provide a conversion translator from the object language of section 4.2 to the language of section 5.2. This conversion translator is written in the translation language of section 4.1, hence its translation is executable on the interpreter of section 4.3.

```
BEGIN GRAMMAR
G = ["] L R G
  | ["] L R
  ;
R = [:"] ':' L
  | [&] '&' R R
  | [|] '|' R R
  | [>] '>' L
  | ["] '"' L
  ;
L = 'A' [A] | 'B' [B] | 'C' [C] | 'D' [D] | 'E' [E] | 'F' [F]
  | 'G' [G] | 'H' [H] | 'I' [I] | 'J' [J] | 'K' [K] | 'L' [L]
  | 'M' [M] | 'N' [N] | 'O' [O] | 'P' [P] | 'Q' [Q] | 'R' [R]
  | 'S' [S] | 'T' [T] | 'U' [U] | 'V' [V] | 'W' [W] | 'X' [X]
  | 'Y' [Y] | 'Z' [Z] | '=' [=] | ';' [;] | '(' [(] | ')' [)]
  | '|' [|] | '&' [&] | ':' [:] | '>' [>] | '"' ["] | ' ' [ ]
  | '[' [[] | ']' []] | '*' [*] | ''' [']
  ;
END GRAMMAR
```

5.3 The Extended Translation Interpreter

A new interpreter is defined that is an extension of the old, and will implement the extended object language.

An additional function to compute the string that would be recognized by a specific rule is named Literal and has the following functionality:

```
Literal: STRING(i) -> STRING
```

All other functions have the same functionality and purpose as in the original machine.

The Functional Definition of the Extended Machine:

```
Machine (G,I) =
 IF Test (G,Skip(G),I) AND equal (Remaining (G,Skip(G),I),NULL)
    THEN
       (TRUE,Emit (G,Skip(G),I)
    ELSE
       (FALSE,NULL)
END Machine
```

```
Test (G,R,I) =
  CASE First (R) OF
  ':' : Test (G,Find(G,rest(R)),I)
  '&' : IF Test (G,rest(R),I)
           THEN
              Test (G,Skip(rest(R)),Remaining(G,rest(R),I))
           ELSE
              FALSE
  '|' : IF Test (G,rest(R),I)
           THEN
              TRUE
           ELSE
              Test (G,Skip(rest(R)),I)
  '>' : TRUE
  '"' : equal (first(rest(R)),first(I))
  '*' : TRUE
  END CASE
END Test

Remaining (G,R,I) =
  CASE first(R) OF
  ':' : Remaining (G,Find(G,rest(R)),I)
  '&' : Remaining (G,Skip(rest(R)),Remaining(G,rest(R),I))
  '|' : IF Test(G,rest(R),I)
           THEN
              Remaining(G,rest(R),I)
           ELSE
              Remaining(G,Skip(rest(R)),I)
  '>' : I
  '"' : rest(I)
  '*' : IF Test (G,rest(R),I)
           THEN
              Remaining (G,R,Remaining (G,rest(R),I) )
           ELSE
              I
  END CASE
END Remaining

Emit (G,R,I) =
 CASE First(R) OF
 ':' : Emit (G,Find(G,rest(R)),I)
 '&' : concat(Emit(G,rest(R),I),Emit(G,Skip(rest(R)),Remaining
          (G,rest(R),I)))
 '|' : IF Test (G,rest(R),I)
          THEN
             Emit (G,rest(R),I)
          ELSE
             Emit (G,Skip(rest(R)),I)
 '>' : first(rest(R))
 '"' : NULL
 '*' : IF Test (G,rest(R),I)
          THEN
             concat (Emit(G,rest(R),I), Emit(G,R,Remaining(G,rest(R),I))
          ELSE
             NULL
 END CASE
END Emit
```

```
Skip (R) =
  CASE first (R) OF
  ':' : Skip(rest(R))
  '&' : Skip(Skip(rest(R)))
  '|' : Skip(Skip(rest(R)))
  '>' : rest(rest(R))
  '"' : rest(rest(R))
  '*' : Skip(rest(R))
  END CASE
END Skip

Find (G,R) =
  IF equal (Literal(G),Literal(R))
     THEN
        Skip(G)
     ELSE
        Find (Skip(Skip(G)),R)
END Find

Literal (R) =
  CASE first(R) OF
  '&' : concat (Literal(rest(R)),Literal(Skip(rest(R))))
  '"' : first(rest(R))
  END CASE
END Literal
```

6. A Minimal Recognizer

The grammars and interpreters presented in this paper are a result of extending a much simpler grammar and interpreter. The initial self-describing grammar and compatible interpreter were designed as an answer to the question, what is the simplest mechanism necessary to implement the recognition of interesting languages? A simple Metalanguage is proposed. The grammar which defines this language is self-describing and is interesting because it is extremely concise.

This class of grammars can be formalized as a triple, G = (V_t,S,P) where:

V_t is a finite set of symbols called terminals.
S is a distinguished symbol not in V_t.
The union of V_t and S will be called V.

P is a finite set of productions, where a production p is a finite sequence of symbols in V+. For all p in P find u, v elements of V+. u is derivable from v if u can be created by the substitution of p for any occurrence of S in v or any derivation of v.

The productions of P can be represented as $p_1 \mid p_2 \mid p_3 \mid \ldots \mid p_n$. The same implementation restrictions apply to this class of recognition grammars as apply to the translation grammars of section 3.

6.1 An Expression Language for Recognizer Description

A translator defined in the language of section 4.1 defines an expression language for this class of grammars, as well as the translation of this language to the object language executable by the interpreter, defined in section 6.3. The symbol '.' will be used for S.

```
BEGIN GRAMMAR
G = A ';' ;
A = [|] C '|' A
  | C
  ;
C = [&] N C
  | N
  ;
N = ["] ''' L '''
  | '(' A ')'
  | [.] '.'
  ;
L = '&' > [&] | '|'   [|] | '"'   ["] | '.'   [.] ;
END GRAMMAR
```

6.2 The Recognizer Object Language

The object language produced by the translator described in section 6.1 is described in the metatranslation language of section 4.1. This object language is only capable of describing recognizers.

```
BEGIN GRAMMAR
S = '.'
  | '&' S S
  | '|' S S
  | ''' ('&'|'|'|'''|'.')
  ;
END GRAMMAR
```

This same recognizer is described in the syntax of the metatranslator described in section 6.1.

```
'.'|'&'..|'|'..|'''('&'|'|'|'''|'.');
```

The translation of the above recognizer by the metatranslator of section 6.1 is presented. It is in the object language executable by the interpreter of section 6.3. As it is a recognizer for this object language it will recognize its own description. That is, this object language program is a self-describing metalanguage that is also capable of the description of any language describable by the metatranslator of section 6.1.
(paragraphing added)

```
|''.
|&''&
  &.
 |&''|
   &.
  &'''
    |''&
     |''|
      |''''
       ''.
```

6.3 A Recognition Interpreter

This interpreter is a much simpler version of the original interpreter presented in section 4.3. The Emit function has been removed and the functionality of Machine has changed to:

Machine: GRAMMAR x INPUT -> RECOGNIZE

The definition of A Recognition Machine:

```
Machine (G,I) =
  IF Test (G,G,I) AND equal(Remaining (G,G,I) = NULL)
     THEN
        TRUE
     ELSE
        FALSE
END Machine

Test (G,R,I) =
 CASE first (R) OF
  '.' : Test (G,G,I)
  '&' : IF Test (G,rest(G),I)
          THEN
             Test (G,Skip(rest(R)),Remaining(G,rest(R),I))
          ELSE
             FALSE
  '|' : IF Test (G,rest(R),I)
          THEN
             TRUE
          ELSE
             Rest (G,Skip(rest(R)),I)
  '"' : equal (first(rest(R)),first(I))
 END CASE
END Test

Remaining (G,R,I) =
 CASE first (R) OF
  '.' : Remaining (G,G,I)
  '&' : Remaining (G,Skip(rest(R)),Remaining(G,rest(R),I))
  '|' : IF Test(G,rest(R),I)
          THEN
             Remaining(G,rest(R),I)
          ELSE
             Remaining(G,Skip(rest(R)),I)
  '"' : rest(I)
 END CASE
END Remaining
```

```
Skip (R) =
  CASE first (R) OF
   '.' : rest (R)
   '&' : Skip(Skip(rest(R)))
   '|' : Skip(Skip(rest(R)))
   '"' : rest(rest(R))
  END CASE
END Skip
```

7. Conclusions

A class of grammars has been defined for which a translator can be concisely stated and simply implemented. This class of grammars is sufficiently powerful to allow the definition of more expressive languages. Although the definition of the translation interpreter is by no means efficient, more practical implementations with equivalent functional properties have been conceived. Efficiency is of minor concern because the primary reason to create a very simple translation system is the construction of intermediate tools for the fabrication of some specific translator.

An interesting language for which to create a translator and corresponding interpreter would be a language similar to the recursive algorithmic notation used to describe the translation interpreters. Once this is done, extensions to the translator necessitating modifications to the interpreter could be more easily implemented.

Techniques to facilitate the creation of powerful problem oriented languages will continue to be investigated. Limiting the problem to finding the smallest useful yet implementable system has provided several important insights, as well as a possibly fertile seed for the future "evolution" of a sophisticated translator writing system.

Acknowledgements:

The results presented in this paper are the product of my involvement with Bill McKeeman's research group on "Zen and the Art of Translator Implementation". The group meetings and discussions with individual members were invaluable to the formulation and refinement of this material. Many other UCSC faculty members and students provided helpful insights and suggestions. I would like to express particular gratitude to Bill McKeeman, Jim Horning, Frank DeRemer, Frank Frazier, Bill Fitler and Dan Ross for all of their time and energy. I would also like to thank the Information Science Department at UCSC for providing an environment conducive to individual research at an undergraduate level.

References:

[Chomsky 57] Chomsky, Norm, Syntactic Structures, Mouton and Co., The Hague, The Netherlands (1957).

[DeRemer 71] DeRemer, F. L., "Simple LR(k) grammars", Comm. ACM, 14, No. 7, 453-460 (1971).

[Floyd 63] Floyd, R. W., "Syntactic Analysis and Operator Precedence," JACM, 10, No. 3, 316-333 (1963).

[Hopcroft 69] Hopcroft, J. E., and Ullman, J. D., Formal Languages and Their Relation to Automata, Addison-Wesley, Reading, Mass. (1969).

[Knuth 65] Knuth, D. E., "On the Translation of Languages from Left to Right," Information and Control, 8:6, 607-639, (1965).

[McKeeman 76] McKeeman, W. M., Private Communication, (1976).

[Schorre 64] Schorre, D. V., "A Syntax Oriented Compiler Writing Language," 1964 ACM National Conference (1964).

[Wirth 66] Wirth, N., and Weber, H. "Euler: A Generalization of Algol 60 and Its Formal Definition," CACM, 9:1, 9:2, (1966).

[Wozencraft 69] Wozencraft, J. M., and Evans, A., Jr. Notes on Program Linguistics, Dept. of Elec. Eng., Mass. Inst. of Tech., Cambridge, Mass., (1969).

OFFICIAL DISTRIBUTION LIST

Contract N00014-76-C-0682

Defense Documentation Center
Cameron Station
Alexandria, VA 22314
12 copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 copies

Office of Naval Research
Code 102IP
Arlington, VA 22217
6 copies

Office of Naval Research
Code 200
Arlington, VA 22217
1 copy

Office of Naval Research
Code 455
Arlington, VA 22217
1 copy

Office of Naval Research
Code 458
Arlington, VA 22217
1 copy

Office of Naval Research
Branch Office, Boston
495 Summer Street
Boston, MA 02210
1 copy

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, IL 60605
1 copy

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106
1 copy

New York Area Office
715 Broadway - 5th Floor
New York, NY 10003
1 copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, DC 20375
6 copies

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps (CodeRD-
Washington, D. C. 20380
1 copy

Naval Electronics Laboratory Center
Advanced Software Technology Division
Code 5200
San Diego, CA 92152
1 copy

Mr. E. H. Gleissner
Naval Ship Research & Development Cent
Computation and Mathematics Department
Bethesda, MD 20084
1 copy

Captain Grace M. Hopper
NAICOM/MIS Planning Branch (OP-916D)
Office of Chief of Naval Operations
Washington, D. C. 20350
1 copy

Mr. Kin B. Thompson
Technical Director
Information Systems Division (OP-911G)
Office of Chief of Naval Operations
Washington, D. C. 20350
1 copy